Hardware-based speculation

Exploiting more instruction-level parallelism requires that we overcome the limitation of control dependences. Overcoming control dependences is done by **speculating on the outcome of branches** and executing the program **as if our guesses were correct**. This mechanism presents an extension over branch prediction with dynamic scheduling. Of course, we need mechanisms to handle the situation where the speculation is incorrect.

Hardware-based speculation combines three key ideas:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow the execution of instructions before the control dependences are resolved—with the ability to undo the effects of an incorrectly speculated sequence
3. Dynamic scheduling to deal with the scheduling of different combinations of basic blocks

Hardware-based speculation follows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a *data flow execution*: operations execute as soon as their operands are available.

To extend Tomasulo's algorithm to support speculation, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative.

Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence *instruction commit*.

The key idea behind implementing speculation is to allow instructions to execute out of order but to **force them to commit in order** and to prevent any irrevocable action until an instruction commits.

The reorder buffer

Adding the commit phase to the instruction execution sequence requires an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer is called *reorder buffer* (ROB). The reorder buffer provides additional registers in the same way as the reservation stations in Tomasulo's algorithm extend the register set: the ROB is a source of operands for instructions between when they execute and when they commit.

Each entry in the ROB contains four fields:

Field	Description
Instruction type	Indicates whether the instruction is a branch, a store or a register operation.
Destination	Supplies the register number (for loads and ALU operations) or the memory address (for stores).
Value	Holds the value of the instruction result until the instruction commits.
Ready	Indicates that the instruction has completed execution and the value is ready.

Although the renaming function of the reservation stations is replaced by the ROB, we still need a place to buffer operations and operands between the time they issue and the time they begin execution. This function is still provided by the reservation stations.

Instruction execution steps

Here are the four steps involved in instruction execution:

Step	Description
Issue	Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and empty slot in the ROB; send the operands to the reservation station if they are available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservation stations are full or the ROB is full, then instruction issue is stalled until both have available entries.
Execute	If one or more operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution for a store at this point is only effective address calculation.
Write result	When a result is available, write it on the CDB and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions: if the value

Step	Description
	to be stored is not available yet, the CDB must be monitored until a value is broadcast, at which time the value field of the ROB entry of the store is updated.
Commit	There are three different sequences of actions at commit, depending on whether the committing instructions is a branch with an incorrect prediction, a store or any other instruction. We see these in detail below.

The commit phase

Depending on the instruction to be committed, we have three possible commit phases:

1. **Normal commit**: the normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB.
2. **Store instruction commit**: committing a store instruction is similar to the normal commit, except that memory is updated rather than a result register.
3. **Incorrect prediction commit**: when a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and the execution is restarted at the correct successor of the branch.

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry.

Speculation vs. no speculation

One of the main differences between a speculative processor and a non-speculative one is that a processor with a reorder buffer can dynamically execute code while maintaining a precise interrupt model, since instructions are **forced to commit in order**.

Furthermore, because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted.

In practice, processors that speculate try to recover as early as possible after a branch is mispredicted. The recovery process is as simple as clearing the ROB for all entries that appear after the mispredicted branch and restarting the fetch at the correct branch successor.

Exception handling

In speculative processors, exceptions are handled by not recognising the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. This allows for a precise exception handling scheme in speculative processors.

Avoiding hazards

WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB and, hence, no earlier loads or stores can still be pending.

RAW hazards through memory are maintained by two restrictions:

1. Not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a destination field that matches the value of the A field of the load.
2. Maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data.